TIM

For a binary search tree, the order of inserting values affects the shape of the tree.
Our goal with binary search trees is to minimize their height.
The ideal binary search tree is complete, where each node above the last level has 2 subnodes, but that is hard to maintain.
An AVL tree is an approximately balanced binary search tree that maintains a balance factor value in each node to ensure balance.
A node is considered imbalanced if the absolute value of the difference in the height of its left subtree and the height of its right subtree is greater than 1.
A node that is imbalanced is considered the node of imbalance, and can be represented by alpha.
There are four potential cases where an AVL tree can become unbalanced.
For left-left imbalance (case 1), the height of alpha's left subtree is larger than the height of its right subtree, and the most recently added node was to the left of its parent.
For left-right imbalance (case 2), the height of alpha's left subtree is larger than the height of its right subtree, and the most recently added node was to the right of its parent.
For right-left imbalance (case 3), the height of alpha's right subtree is larger than the height of its left subtree, and the most recently added node was to the left of its parent.
For right-right imbalance (case 4), the height of alpha's right subtree is larger than the height of its left subtree, and the most recently added node was to the right of its parent.
Cases 1 and 4 and cases 2 and 3 are mirrors of each other.
For case 1 and case 2, we call alpha's left child node A, while for case 3 and 4, we call alpha's right child node A.
For case 2, we call A's right child node B, and for case 3, we call A's left child node B.
To rebalance case 1, we perform a single rotation by setting A's right child to be alpha and moving A's old right subtree to be the left subtree of alpha
To rebalance case 4, we perform a single rotation by setting A's left child to be alpha and moving A's old left subtree to be the right subtree of alpha
To rebalance case 2, we first perform a single rotation with nodes A and B by setting alpha's new left child to be node B, setting node B's new left child to be node A, and by setting A's new right child to be node B's old left subtree.
Once we perform the first rotation for case 2, the tree becomes case 1, where the old node B can be considered node A, while alpha remains alpha; the tree can be balanced using the same rotation needed to case 1.
To rebalance case 3, we first perform a single rotation with nodes A and B by setting alpha's new right child to be node B, setting node B's new right child to be node A, and by setting A's new left child to be node B's old right subtree.
Once we perform the first rotation for case 3, the tree becomes case 4, where the old node B can be considered node A, while alpha remains alpha; the tree can be balanced using the same rotation needed to case 4.
When you insert a node into an AVL tree, you should update the heights on the path from the newly inserted node to the root node of the tree, checking for imbalance as you go.
Computer memory can be stored in a number of different locations; in descending order of speed, they are the CPU, registers, the L1 cache, the L2 cache, RAM, and SDD/HDD.
SDD/HDD is multiple orders of magnitude slower than RAM, but have lots of storage and are persistent.
The goal of database systems is to minimize the number of SDD/HDD accesses since a worst case binary search search is significantly faster than a single additional access of SDD/HDD data. Raw binary search trees are not good for this because a given node of the binary tree probably occupies only a fraction of any cache line. B-trees are a way to get better locality by putting multiple elements into each tree node. B-trees were originally invented for storing data structures on disk, where locality is even more crucial than with memory.

Accessing a disk location takes about 5ms = 5,000,000ns. Therefore, if you are storing a tree on disk, you want to make sure that a given disk read is as effective as possible. B-trees have a high branching factor, much larger than 2, which ensures that few disk reads are needed to navigate to the place where data is stored. B-trees may also be useful for in-memory data structures because these days main memory is almost as slow relative to the processor as disk drives were to main memory when B-trees were first introduced! A B-tree of order m is a search tree in which each non-leaf node has up to m children. The actual elements of the collection are stored in the leaves of the tree, and the non-leaf nodes contain only keys. Each leaf stores some number of elements; the maximum number may be greater or (typically) less than m. The data structure satisfies several requirements: First, every path from the root to a leaf has the same length. Second, if a node has n children, it contains n−1 keys. Third, every node (except the root) is at least half full. Fourth, the elements stored in a given subtree all have keys that are between the keys in the parent node on either side of the subtree pointer. (This generalizes the BST invariant.) Fifth, the root has at least two children if it is not a leaf. Because the height of the tree is uniformly the same and every node is at least half full, we are guaranteed that the asymptotic performance is O(log n) where n is the size of the collection. The real win is in the constant factors, of course. We can choose m so that the pointers to the m children plus the m−1 elements fill out a cache line at the highest level of the memory hierarchy where we can expect to get cache hits. For example, if we are accessing a large disk database then our "cache lines" are memory blocks of the size that are read from disk. Lookup in a B-tree is straightforward. Given a node to start from, we use a simple linear or binary search to find whether the desired element is in the node, or if not, which child pointer to follow from the current node. Insertion and deletion from a B-tree are more complicated. For insertion, we first find the appropriate leaf node into which the inserted element falls (assuming it is not already in the tree). If there is already room in the node, the new element can be inserted simply. Otherwise the current leaf is already full and must be split into two leaves, one of which acquires the new element. The parent is then updated to contain a new key and child pointer. If the parent is already full, the process ripples upwards, eventually possibly reaching the root. If the root is split into two, then a new root is created with just two children, increasing the height of the tree by one. Deletion works in the opposite way: the element is removed from the leaf. If the leaf becomes empty, a key is removed from the parent node. If that node is no longer at least half full, the keys of the parent node and its immediate right (or left) sibling are reapportioned among them so that invariant 3 is satisfied. If this is not possible, the parent node can be combined with that sibling, removing a key another level up in the tree and possibly causing a ripple all the way to the root. If the root has just two children, and they are combined, then the root is deleted and the new combined node becomes the root of the tree, reducing the height of the tree by one.
Hash tables effectively function like a Python dictionary.
The hash table itself is a 1-dimensional array. Each index stores a list, which contains the key value pairs as individual tuples.
The size of a hash table refers to how large the array is. The size of any hash table is denoted by the letter m.
The load factor of a hash table is the number of inserted elements divided by the size of the hash table. The load factor of any hash table is denoted by the greek letter lambda.
Inserting into a hash table works by passing the key of a key-value pair into a hashing function. The inner mechanism of the hash function is not very important, but it typically returns the value mod the table size, so that the output is a valid index in the table. The key-value pair is then appended as a tuple to the end of the list stored at the index outputted by the hash function.
Looking up a key in a hash table works by passing the desired key through the hash function, accessing all the tuples in the list stored at the corresponding index of the hash table, and performing a linear search to match the searched-for key with the key in each key-value pair.
We want to maintain a low load factor such that lambda is not greater than 0.9. If lambda becomes larger than 0.9, we should rehash the table, which means using a larger hash table (with more indices), updating the hash function accordingly,

and passing all of the stored values through the new hash function and re-storing them accordingly.

The relational data model has several benefits. First, it is for the most part the standard data model and query language used in industry. Second, the model is ACID compliant, meaning that it adheres to the principles of atomicity, consistency, isolation, and durability. Third, the relational model works well with highly structured data. Fourth, the relational model can handle large amounts of data. Finally, the relational model is well understood with lots of tooling and lots of experience.
A transaction is a series of CRUD operations performed as a single unit of work. CRUD stands for Create, Read, Update, Delete, and refer to the main operations you can perform on a database. A transaction could be as simple as a SELECT statement, or more involved like joint update statements, etc. An entire transaction either succeeds (COMMIT) or fails (ROLLBACK/ABORT).
The four ACID properties are atomicity, consistency, isolation, and durability. Atomicity means that every transaction is treated as one unit; the unit is either executed in full or no parts of it are executed. Consistency means that any transaction takes the database from one consistent state to another. A consistent state is when all the data meets integrity conditions. Isolation means that any two transactions cannot affect each other if executed at the same time. An issue with isolation could only happen if one transaction is reading from the data that another transaction is writing. A dirty read is when the first transaction is able to read a row that has been written by the second transaction, but has not yet been committed. A non-repeatable read is when a transaction returns different data than what existed when it started executing. Finally, phantom reads occur when a transaction reads data twice and inserts both instances instead of just once. Durability means that once a transaction is completed and committed, the changes are permanent.

Shrek 5 is set to hit theaters on 12/23/2026.

To increase the capacity of a system, you can scale either vertically or horizontally. Scaling vertically refers to adding more compute within one system. Scaling horizontally means adding more systems with the same compute.
A distributed system is a collection of independent computers that appear to its users as one computer. The main characteristics of distributed systems is that they operate concurrently, fail independently, and do not have a global clock. It is inevitable that distributed systems will need to have network partitioning, which means that the systems need to be partition tolerant. In other words, if something happens to one node, the system as a whole will be fine.
The CAP theorem has three parts: consistency, availability, and partition tolerance. The theorem states that you can always have two of the three, but never all three. Consistency means that you will always get the same result from the system. Availability means that you can always access the system. Partition tolerance means that the system will continue to operate despite network issues. Examples of products that include consistency and availability but not partition tolerance are relational database models like PostgreSQL and MySQL. Examples of products that include consistency and partition tolerance but not availability are mongo, redis, and hbase. Examples of products that include availability and partition tolerance but not consistency are couchDB, cassandra, and dynamoDB.

NIDHI
Foundations & Searching
Searching is the most common operation performed by a database system. In SQL, the SELECT statement is arguably the most versatile or complex because you can nest the statement. The baseline for efficiency is Linear Search: Start at the beginning of a list and proceed element by element until you find what you’re looking for or you get to the last element and have not found it. If there are n elements, worst case scenario is going through all n values.
A record is a collection of values for attributes of a single entity instance (a

row of a table). A collection is a set of records of the same entity type (a table). Some collections are stored sequentially like a list. Search Key is a value for an attribute from the entity type, could be more than 1 attribute. If each record takes up x bytes of memory, then for n records, we need n*x bytes of memory. There are 2 ways of storing the n*x bytes in RAM memory: Contiguously allocated list where all n*x bytes are in a single chunk of memory, and Linked list, where there are x + 2 memory addresses (that identify front versus back) * n. For a linked list, each record needs x bytes + additional space for 1 or 2 memory addresses and individual records are linked together in a type of chain using memory addresses. Python technically does not have a true equivalent to a contiguous array. Arrays are fast for random access but slow for random insertions (anywhere but the end). Linked Lists are slow for random access but fast for random insertions.
Binary search keeps going to the middle iteratively/recursively, but it must be a sorted array. The input is an array of values in sorted order, target value. Output is the location (index) of where target is located or some value indicating target was not found. The maximum number of searches = log base 2 of (n). Binary search only really applies to contiguously allocated list because it is super inefficient for linked lists. Best case is the target is found at middle; 1 comparison (inside the loop). Worst case is the target is not in the array; log2 n comparisons, or O(log2n) time complexity.
For linear Search, Best case is the target is found at the first element; only 1 comparison, and Worst case is the target is not in the array; n comparisons (O(n) time complexity).
Let’s say you have a table with two columns: id and specialVal. Assume data is stored on disk by the id column’s value. Searching for a specific id would be fast, but if we want to search for a specific specialVal, the only option is linear scan of that column. We cannot store data on disk sorted by both id and specialVal at the same time, because the data would have to be duplicated which is space inefficient. Therefore, we need an external data structure to support faster searching by specialVal than a linear scan. One of your current options is an array of tuples (specialVal, rowNumber) sorted by specialVal, and we could use Binary Search to quickly locate a particular specialVal and find its corresponding row in the table. But every insert into the table would be like inserting into a sorted array, which is slow. Another option is a linked list of tuples (specialVal, rowNumber) sorted by specialVal, but searching for a specialVal would be slow, since a linear scan required. But inserting into the table would theoretically be quick to also add to the list. Something with Fast Insert and Fast Search that would solve this issue is a Binary Search Tree where every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent.
Extra notes about binary search and AVL trees:
https://www.cs.rochester.edu/u/gildea/csc282/slides/C12-bst.pdf
ttps://ics.uci.edu/~thornton/ics46/Notes/AVLTrees/
Creating/inserting into a binary search tree conceptually
Let’s say you want to insert: 23 17 20 42 31 50. 23 is the top node (root), 17 will descend to the left because it is smaller, 20 will descend to the right of 17, 43 will descend to the left from 23, 31 will go left, 50 will go right from 43. The level order traversal of this tree would be 23 17 43 20 31 50.
Tree traversal types are preorder, post order, In order, and Level order.
How? When processing 23 (root), print 23 and then temporarily store 17 and 43 in an external data structure. Then, print 17 and remove it from the external data structure and temporarily store 20. Then, remove 43 from the external data and print, and temporarily store 31 and 50. Then remove 20 and print, no child so prints 31 then print 50.
Temporarily storing data happens in a queue, and Python has a special version called a deque.
Binary Search tree in python:
CLASS Binary tree node(self, value, left, right)
Value - integer
Left – binary tree node
right – binary tree node

Root = binarytreenode(23)

Root.left = binarytreenode(17)
Root.right= binarytreenode(43)
How to get 20?:
Root.left.right = binarytreenode(20)

The Order of values inserted into binary search tree matters (changes the shape of the tree). For example, If the values are sorted, it would be an unbalanced tree. The goal is to minimize height of tree. Minimum height is always going to be from a complete tree (all nodes filled except for last level). An AVL tree is an approximately balanced binary search tree, it is Self balancing and Maintains a balance factor in each node (at that node, how balanced is the tree if that node were the root). A tree is an AVL TREE IF the absolute value of the height of left sub tree minus the height of right sub tree is less than or equal to 1. Basically height of left sub tree and right sub tree cannot differ by more than 1. Inserting into an AVL tree can balance or imbalance the tree, but it would only change on the path from the most recently inserted node to the root. Once a tree becomes imbalanced, there is an algorithm that will rebalance the tree. Alpha (root where it is imbalanced) node gets reorganized. Inserting can cause an imbalance in 4 ways:
* Left left insertion: when you insert to the left subtree of the left child of the node of imbalance
* Left right case: when you insert into the right subtree of the left child of the node of imbalance
* Right left case: insert into the left subtree of the right child
* Right right: insert into the right subtree of the right child
Rebalancing:
* Rebalancing case 1: single rotation (also applies to case 4), reassign c’s left pointer to point to t2 and reassign a’s right pointer to point to c
* Rebalancing case 2: double rotation (also applies to case 3). Need to separate t2 into its 2 children and basically do a single rotation twice
How the reassignment of pointers happens:
* Case 4 (RR):
* A.right points to the left child of C
* C.left points to A
Calculating height of a node is the maximum of the 2 heights of a node's subtrees, then you add one to the larger height.
MEMORY:
A CPU has 16 or 32 registers, depending on the processor. Registers are the closest to the CPU, making them very fast but small and expensive. The L1 cache is larger than registers but slower and cheaper per byte. The L2 cache is even larger than L1. RAM is the primary memory and operates at nanosecond speed. Secondary storage, such as SSDs and HDDs, is much slower, measured in milliseconds. SSDs and HDDs provide a lot of storage and are persistent, meaning they can survive power cycles, but they are extremely slow. To improve speed, database systems should minimize access to secondary storage. A 64-bit integer takes up 8 bytes of memory.
Let’s say there is an AVL tree where each node has a key and a value, and then a left and right pointers. On a 64 bit machine, this would take 4 * 8 = 32 bytes (assuming an integer takes 8 bytes) BUT if each node is stored in a different block, each node would need 2048 bytes. For an AVL tree, 1.44 log base 2 of n, where n is the number of nodes in the tree, is worst case.
Let’s say you have a sorted array of 128 integers. Worst case binary search on 128 integers is WAY faster than a single additional disk access. Even faster would be to do 3 children per node (shallower/less tall tree). This would minimize secondary storage disk accesses, which is the most important thing for database systems.

B+ Tree
B+ Tree is designed to maximize the number of values stored in each disk block. Each node has the maximum possible number of keys to reduce disk access. A node with N-1 keys has N children. The B+ Tree is optimized for disk-based indexing by minimizing disk access. It is an m-way tree with order M, where M is the maximum number of keys in a node, and M+1 is the maximum number of children. All nodes, except the root, must be at least half full, but the root does not have

this requirement. Insertions always happen at the leaf level, and leaves are stored as a doubly linked list. Keys in nodes are kept sorted. A B+ Tree is shallower and wider than a BST or AVL tree with the same number of nodes. There are two types of nodes: internal nodes, which store keys and pointers to children, and leaf nodes, which store keys and data. Unlike B-Trees, which are used for in-memory indexing, B+ Trees are designed for disk-based indexing.

A relational database management system (RDBMS) increases efficiency through indexing, direct storage control, column-oriented or row-oriented storage (with column-oriented being faster for some large data processing), materialized views, query optimization, caching, prefetching, precompiled stored procedures, and data replication and partitioning to improve performance and data management.

Column vs row-oriented storage
Column-oriented storage and row-oriented storage have different advantages depending on the use case. Transactions must be fully completed or not executed at all, following a commit or rollback/abort process. This ensures data integrity, error recovery, concurrency control, reliable data storage, and simplified error handling. The relational model offers several benefits, including a mostly standard data model and query language, ACID compliance (Atomicity, Consistency, Isolation, Durability), strong support for highly structured data, the ability to handle large amounts of data, and widespread understanding with extensive tooling and experience.

ACID:
ACID ensures reliable database transactions through four key properties: Atomicity, meaning a transaction must be fully completed or not executed at all (all or none); Consistency, transaction is always in a consistent state (all data meets integrity constraints); Isolation, preventing one transaction from negatively affecting another, often managed through locking to avoid issues like dirty reads (where a transaction reads uncommitted changes from another), non-repeatable reads (where repeated queries within a transaction return different results due to committed changes by another transaction), and phantom reads (where rows are added or deleted by another transaction while the first is still running); and Durability, guaranteeing that once a transaction is committed, its changes are permanent even in the event of a system failure.
Relational databases have downsides, such as schemas changing over time, not all apps needing full ACID compliance, joins being expensive, and a lot of data being semi-structured or unstructured like JSON or XML. Horizontal scaling can be hard, and some apps need something more performant, like real-time or low-latency systems.

Scaling:
Conventional Wisdom is to scale vertically (up, with bigger, more powerful systems) until the demands of high-availability make it necessary to scale out with some type of distributed computing model. Why? Because scaling up is easier since there is no need to really modify your architecture. But, there are practical and financial limits to this. However, there are modern systems that make horizontal scaling less problematic.
One such solution is Distributed systems. A distributed system is “a collection of independent computers that appear to its users as one computer.” -Andrew Tennenbaum
Characteristics of Distributed Systems are that computers operate concurrently, computers fail independently, no shared global clock. Distributed storage has 2 directions: replication and sharding. Replication is having the same data in multiple places. Sharding is having the data broken up into groups. Data is stored on more than 1 node, typically replicated (each block of data is available on N nodes). Distributed databases can be relational or non-relational. MySQL and PostgreSQL support replication and sharding. CockroachDB is a new player on the scene. Many NoSQL systems support one or both models. Network partitioning is inevitable (network failures, system failures), so overall system needs to be Partition Tolerant, in other words, the system can keep running even with network partition.

The CAP Theorem states it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees: consistency, availability, and partition tolerance. Consistency means that every read receives the most recent write or error thrown. Availability means that every request receives a (non-error) response - but no guarantee that the response contains the most recent write. Partition tolerance means that the system can continue to operate despite arbitrary network issues.
CAP theorem applied to a database (think of it like a 3 way venn diagram). Consistency means that every user of the DB has an identical view of the data at any given instant (the definition of Consistency in CAP is different from that of ACID.) Availability means that in the event of a failure, the database remains operational. Partition Tolerance means the database can maintain operations in the event of the network's failing between two segments of the distributed system.
Consistency and Availability means the system always responds with the latest data and every request gets a response, but may not be able to deal with network issues.
Consistency and Partition Tolerance means if system responds with data from a distributed store, it is always the latest, else data request is dropped.
Availability and Partition Tolerance means the system always sends are responds based on distributed store but may not be the absolute latest data.
In reality, CAP theorem says if you cannot limit the number of faults, requests can be directed to any server, and you insist on serving every request, then you cannot possibly be consistent. But it is interpreted as you must always give up something: consistency, availability, or tolerance to failure.

NoSQL and Key Value Databases
ACID transactions focus on "data safety" and are considered a pessimistic concurrency model because it assumes one transaction must protect itself from other transactions. AKA it assumes that if something can go wrong, it will. Conflicts are prevented by locking resources until a transaction is complete (there are both read and write locks). The Write Lock Analogy says that it's like borrowing a book from a library, if you have it, no one else can.
Optimistic Concurrency says that transactions do not obtain locks on data when they read or write. It is considered optimistic because it assumes conflicts are unlikely to occur. Even if there is a conflict, everything will still be OK because you add last update timestamp and version number columns to every table and read them when changing. THEN, check at the end of transaction to see if any other transaction has caused them to be modified. Low Conflict Systems (backups, analytical databases, etc.) are read heavy systems. the conflicts that arise can be handled by rolling back and re-running a transaction that notices a conflict, so, optimistic concurrency works well - allows for higher concurrency. High Conflict Systems work by rolling back and rerunning transactions that encounter a conflict, which is less efficient, so, a locking scheme (pessimistic model) might be preferable.
NoSQL
"NoSQL" first used in 1998 by Carlo Strozzi to describe his relational database system that did not use SQL. More common modern meaning is "Not Only SQL". But, sometimes thought of as non-relational databases. Idea originally developed, in part, as a response to processing unstructured web-based data.
ACID Alternative for Distributed Systems is called BASE. Basically Available guarantees the availability of the data (per CAP), but response can be "failure"/"unreliable" because the data is in an inconsistent or changing state (System appears to work most of the time). Soft State says the state of the system could change over time, even without input. Changes could be result of eventual consistency. Data stores don't have to be write-consistent. Replicas don't have to be mutually consistent. Eventual Consistency means The system will eventually become consistent, and all writes will eventually stop so all nodes/replicas can be updated.
Key Value Databases
key = value
Key-value stores are designed around simplicity, speed, and scalability. Simplicity means the data model is extremely simple, comparatively, tables in a RDBMS are very complex. Simplicity lends itself to simple CRUD ops and API

creation. Speed means it is usually deployed as in-memory DB; retrieving a value given its key is typically a O(1) op because hash tables or similar data structs used under the hood. There is no concept of complex queries or joins because they slow things down. Scalability means horizontal Scaling is simple - add more nodes. Typically concerned with eventual consistency, meaning in a distributed environment, the only guarantee is that all nodes will eventually converge on the same value.
Data science use cases include storing intermediate results from data preprocessing and exploratory data analysis (EDA) or experiment/testing (A/B) results without using the production database, using a feature store for fast retrieval of frequently accessed features for model training and prediction, and model monitoring to store key performance metrics, especially for real-time inference.
Software engineering use cases include storing session information, where all session data can be saved with a single PUT or POST and retrieved quickly with a GET, user profiles and preferences, where user settings like language, time zone, and UI preferences can be fetched with a single GET, shopping cart data, which must be tied to the user and accessible across browsers, machines, and sessions, and using a caching layer in front of a disk-based database for faster access.
Connection pooling does a bunch of stuff, basically makes it more efficient.
Redis:
Redis, or Remote Directory Server, is an open-source, in-memory database that supports durability by saving snapshots to disk at intervals or using an append-only file to track changes for recovery after failures. It is sometimes called a data structure store and is primarily a key-value store, though it also supports models like Graph, Spatial, Full-Text Search, Vector, and Time Series.
Originally developed in 2009 in C++, Redis is extremely fast, handling over 100,000 SET operations per second, and offers a rich collection of commands. However, it does not handle complex data, lacks secondary indexes, and only supports lookups by key.
In Redis, keys are typically strings but can be any binary sequence, while values can be various data types, including strings, lists, hashes, sets, sorted sets, and geospatial data. Redis provides 16 default databases, numbered 0 to 15, and is accessed through commands for setting and retrieving key-value pairs, with many language libraries available. The foundation data type is a string, which maps one string to another and is commonly used for caching frequently accessed HTML/CSS/JS, storing config settings, user info, token management, counting views, or rate limiting.
The hash type stores key-value entries as field-value pairs and is useful for representing objects, session management, user/event tracking, and active session tracking. The list type is a linked list of string values, commonly used for stacks and queues, queue management, logging, social media feeds, chat message history, and batch processing. Linked lists allow efficient O(1) insertion at the front or end. The JSON type supports full JSON syntax and is stored in a binary tree-structure for fast access. The set type is an unordered collection of unique strings, useful for tracking unique items (e.g., IP addresses), primitive relations (e.g., students in a course), access control lists, and social network connections.
Redis-py is the standard client for Python, maintained by Redis itself
Why Redis over MySQL/s3 or other relational database? Because all data is stored in disk, so it is faster. Latency issues (much faster than select statements). Redis pipelines help avoid multiple related calls to the server, which needs less network overhead.

Document Databases and MongoDB
Document Database is a non-relational database that stores data as structured documents, usually in JSON. They are designed to be simple, flexible, and scalable. JSON (JavaScript Object Notation) is a lightweight data-interchange format, easy for humans to read and write, easy for machines to parse and generate. JSON is built on two structures. A collection of name/value pairs (In various languages, this is operationalized as an object, record, struct, dictionary, hash table, keyed list, or associative array) and an ordered list of values (In most languages, this is operationalized as an array, vector, list, or

sequence.) These are two universal data structures supported by virtually all modern programming languages, which makes JSON a great data interchange format.

BSON is binary JSON, it is a binary-encoded serialization of a JSON-like document structure. It supports extended types not part of basic JSON (e.g. Date, BinaryData, etc). It is lightweight and keeps space overhead to a minimum. It is traversable, which means it's designed to be easily traversed, which is vitally important to a document DB. It is efficient because encoding and decoding must be efficient. It is also supported by many modern programming languages.
XML, or extensible markup language, is the precursor to JSON as data exchange format. XML and CSS are used together for web pages content and formatting. XML is structurally like HTML, but tag set is extensible. Some tools/technologies used with XML are Xpath (a syntax for retrieving specific elements from an XML doc), Xquery (a query language for interrogating XML documents; the SQL of XML), DTD (Document Type Definition - a language for describing the allowed structure of an XML document), and XSLT (eXtensible Stylesheet Language Transformation – a tool to transform XML into other formats, including non-XML formats such as HTML.)
Why document databases? They address the impedance mismatch problem between object persistence in OO systems and how relational DBs structure data. OO Programming: Inheritance and Composition of types. How do we save a complex object to a relational database? We basically have to deconstruct it. The structure of a document is self-describing. They are well-aligned with apps that use JSON/XML as a transport layer
MongoDB
Mongodb Started in 2007 after Doubleclick was acquired by Google, and 3 of its veterans realized the limitations of relational databases for serving over 400,000 ads per second. MongoDB was short for Humongous Database. MongoDB Atlas released in 2016, and provided documentdb as a service. No predefined schema for documents is needed. Every document in a collection could have different data/schema. Rich Query Support provides robust support for all CRUD ops. Indexing supports primary and secondary indices on document fields. Replication supports replica sets with automatic failover, and load balancing is built in. MongoDB Atlas is a fully managed MongoDB service in the cloud (DBaaS). MongoDB Enterprise is a subscription-based, self-managed version of MongoDB. MongoDB Community is source-available, free-to-use, self-managed.
Relational database versus Mongo:
A database in a relational database is called a database in MongoDB.
A table or view in a relational database is called a collection in MongoDB.
A row in a relational database is called a document in MongoDB.
A column in a relational database is called a field in MongoDB.
An index in a relational database is called an index in MongoDB.
A join in a relational database is called an embedded document in MongoDB.
A foreign key in a relational database is called a reference in MongoDB.
Interacting with MongoDB is done using the following. mongosh (MongoDB Shell) is a CLI tool for interacting with a MongoDB instance. MongoDB Compass is a free, open-source GUI to work with a MongoDB database. DataGrip and other 3rd Party Tools can also be used. Every major language has a library to interface with MongoDB, such as PyMongo (Python), Mongoose (JavaScript/node), and more. PyMongo is a Python library for interfacing with MongoDB instances.
Intro to graph data model
Graph database is made up of nodes connected by edges, which can be directed or undirected. Each edge and node is uniquely identifiable. You can also add extra properties via key value pairs.
Queries are based on the structure of database, called a cipher. Examples of graph data model are social networks, web pages, chemical and biological data. The underlying thing that links pages of the internet (which is a is big graph) is called https. (404 error message)
A labelled property graph has edges/arcs/relationships and nodes/vertices. This is the type that has properties with keys and values. Labels are used to mark a node as part of a group. Nodes with no relationships are allowed but edges not connected to nodes are not. A path is getting from one node to another in order, with no repeating nodes or edges. It does not matter if you count nodes or edges

in a graph algorithm as long as it is consistent. There are connected (vs. disconnected) graphs, connected means there is a path between any two nodes in the graph. Weighted (vs. Unweighted) means the edge has a weight property (important for some algorithms). Directed (vs. Undirected) means relationships (edges) define a start and end node. Directed graphs can be a loop with arrows going both ways. Acyclic (vs. Cyclic) means the graph contains no cycles. Spare vs dense has to do with the number of edges compared to number of nodes. Trees have no cycles. Pathfinding has to do with finding the shortest path, which can mean fewest edges or lowest weight when summed. Average shortest path can be used to monitor efficiency and resiliency of a network. PageRank search engines used to use the number of links to a website as the first search result, then it became how many pages link to the pages that link to the most page. Breadth first search (visit nearest neighbors first) is different from depth first search (walks down each branch first). Centrality is determining which nodes are “more important” in a network compared to other nodes (degree, closeness, betweenness, pagerank). Community Detection is evaluating clustering or partitioning of nodes of a graph and tendency to strengthen or break apart. Famous graph algorithms include Dijkstra’s Algorithm (single-source shortest path algo for positively weighted graphs), A* Algorithm (similar to Dijkstra’s with added feature of using a heuristic to guide traversal), and PageRank (measures the importance of each node within a graph based on the number of incoming relationships and the importance of the nodes from those incoming relationships).
Neo4j
Neo4j is a NoSQL database that supports both transactional and analytical processing of graph-based data. It is relatively new, schema-optional, ACID-compliant, and supports various types of indexing and distributed computing. Similar databases include Microsoft CosmosDB and Amazon Neptune. Neo4j uses Cypher, a graph query language created in 2011, and supports plugins like APOC for additional functions and the Graph Data Science Plugin for running graph algorithms. Docker Compose is a tool for managing multi-container applications using a YAML configuration file. It allows starting, stopping, and scaling services with a single command, ensuring consistent environments. Interaction is mostly through the command line, and .env files help manage environment variables across platforms. In Neo4j, relationships are directed.

Docker compose:
Docker Compose supports multi-container management with a declarative setup using a YAML file (docker-compose.yaml) to define services, volumes, and networks. A single command can start, stop, or scale multiple services at once, ensuring a consistent environment. Interaction is mostly through the command line, and .env files store environment variables to keep settings separate across platforms (.env.local, .env.dev, .env.prod).
Port numbers
Maximum port number is 65535. Range of ports for system services (need root access) is 0-1023.
http: 80
https: 443
Ssh: 22
Ftp: 21

Retrieval Augmented Generation (RAG)
The outputs are contextual to the input, in other words the pdf of the class notes is contextual elements and augmenting what is already in the model with that context.

ANSLEY

Searching in Database Systems

Searching is a common operation in database systems. In SQL, the SELECT statement is one of the most versatile and complex operations.

Baseline Efficiency: Linear Search

Linear search starts at the beginning of a list and proceeds element by element until:
1. The target is found.
2. The last element is reached without finding the target.

Key Terms

- Record: A collection of attribute values for a single entity instance (a row in a table).
- Collection: A set of records of the same entity type (a table).
- Search Key: A value from an attribute used to search. It can be a single or multiple attributes.

Lists of Records

- Memory Usage: If each record takes x bytes, for n records, the total memory required is n * x bytes.
- Contiguously Allocated List: All n * x bytes are allocated as a single memory block.
- Linked List: Each record takes x bytes, plus additional memory for one or two addresses to link records together.

Contiguous vs. Linked Lists

Contiguous Allocated List (Array)
- Memory is allocated as a single block.
- Faster for random access.
- Slower for insertions, except at the end.

Linked List
- Records are linked by memory addresses.
- Extra storage is needed for the address.
- Faster for insertions anywhere in the list.
- Slower for random access.

Insertion Examples

- Array: Inserting after the second record requires moving 5 records to make space.
- Linked List: Inserting after the second record does not require moving other records.

Observations

- Arrays: Fast for random access, but slow for insertions.
- Linked Lists: Slow for random access, but fast for insertions.

Binary Search

- Input: A sorted array and a target value.
- Output: The index of the target, or an indication that it is not found.

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

- Example: In a sorted array, the target "A" is located by adjusting left or right based on the midpoint comparison.

Time Complexity

- Linear Search:
  - Best case: O(1) (target found at the first element).
  - Worst case: O(n) (target is not found).

- Binary Search:
  - Best case: O(1) (target found at midpoint).
  - Worst case: O(log2 n) (target not found).

Searching in Databases

- Storing by ID: Fast for searching a specific ID.
- Searching by Special Value: Linear scan is required, which is inefficient.

Issue: Storing Data by Both ID and Special Value
- It is not possible to store data sorted by both ID and special value at the same time due to space inefficiency.

Solution: External Data Structures

- Option 1: Array of tuples (specialVal, rowNumber) sorted by specialVal. Binary search can quickly find a specialVal, but inserting is slow because the array is sorted.
- Option 2: Linked list of tuples (specialVal, rowNumber) sorted by specialVal. Insertion is fast, but searching requires a linear scan.

What About Fast Insert and Search?

A Binary Search Tree (BST) can provide both fast insertion and searching. In a BST:
- Each node in the left subtree is less than the parent node.
- Each node in the right subtree is greater than the parent node.

Relational Databases & Distributed Systems
Benefits of the Relational Model

(Mostly) Standard Data Model and Query Language
ACID Compliance
Works well with highly structured data
Can handle large amounts of data
Well understood, extensive tooling and expertise

Relational Database Performance Enhancements

Indexing
Direct storage control
Column-oriented vs row-oriented storage
Query optimization
Caching/prefetching
Materialized views
Precompiled stored procedures
Data replication and partitioning

Transaction Processing
A transaction is a sequence of one or more CRUD operations performed as a single logical unit of work:

Either the entire sequence succeeds (COMMIT)
OR the entire sequence fails (ROLLBACK or ABORT)

Benefits:

Data Integrity
Error Recovery
Concurrency Control
Reliable Data Storage
Simplified Error Handling

ACID Properties
Atomicity

Transaction is treated as an atomic unit
Either fully executed or no parts are executed

Consistency

A transaction takes a database from one consistent state to another
Consistent state: all data meets integrity constraints

Isolation
When two transactions T1 and T2 execute simultaneously, they should not affect each other:

If both T1 and T2 are reading data: no problem
If T1 reads data that T2 may be writing, potential issues include:

Dirty Read: T1 reads a row modified by uncommitted T2
Non-repeatable Read: T1 executes same query twice but gets different values because T2 committed changes
Phantom Reads: T1 is running while T2 adds/deletes rows from the set T1 is using

Durability

Once a transaction completes and commits successfully, changes are permanent
Changes are preserved even during system failure

Example Transaction: Money Transfer
sqlCopyDELIMITER //

```
CREATE PROCEDURE transfer(
    IN sender_id INT,
    IN receiver_id INT,
    IN amount DECIMAL(10,2)
)
BEGIN
    DECLARE rollback_message VARCHAR(255) DEFAULT 'Transaction rolled back: Insufficient funds';
    DECLARE commit_message VARCHAR(255) DEFAULT 'Transaction committed successfully';

    -- Start the transaction
    START TRANSACTION;

    -- Attempt to debit money from sender
    UPDATE accounts SET balance = balance - amount WHERE account_id = sender_id;

    -- Attempt to credit money to receiver
    UPDATE accounts SET balance = balance + amount WHERE account_id = receiver_id;

    -- Check if there are sufficient funds in sender account
    IF (SELECT balance FROM accounts WHERE account_id = sender_id) < 0 THEN
```

```
        -- Roll back the transaction if insufficient funds
        ROLLBACK;
        SIGNAL SQLSTATE '45000'  -- 45000 is unhandled, user-defined error
            SET MESSAGE_TEXT = rollback_message;
    ELSE
        -- Log the transactions if sufficient funds
        INSERT INTO transactions (account_id, amount, transaction_type)
            VALUES (sender_id, -amount, 'WITHDRAWAL');
        INSERT INTO transactions (account_id, amount, transaction_type)
            VALUES (receiver_id, amount, 'DEPOSIT');

        -- Commit the transaction
        COMMIT;
        SELECT commit_message AS 'Result';
    END IF;
END //

DELIMITER ;
```

Limitations of Relational Databases

Schemas evolve over time
Not all applications need full ACID compliance
JOIN operations can be expensive
Semi-structured or unstructured data (JSON, XML) not handled optimally
Horizontal scaling challenges
Performance limitations for real-time, low-latency systems

Scalability: Vertical vs. Horizontal
Conventional Wisdom:

Scale vertically (up) with bigger, more powerful systems
Only scale horizontally (out) when high-availability necessitates distributed computing

Considerations:

Vertical scaling is simpler (no architecture changes) but has practical and financial limits
Modern systems make horizontal scaling more manageable

Distributed Systems
A distributed system is "a collection of independent computers that appear to its users as one computer." -Andrew Tanenbaum
Characteristics:

Computers operate concurrently
Computers fail independently
No shared global clock

Distributed Storage Approaches
Distributed Data Stores

Data stored on multiple nodes, typically replicated
Each block of data is available on N nodes
Can be relational or non-relational:

MySQL and PostgreSQL support replication and sharding
CockroachDB is a newer distributed SQL database
Many NoSQL systems support distribution models

Important Consideration: Network partitioning is inevitable

Network failures, system failures will occur
Overall system needs to be partition tolerant (continue running despite network partitions)

The CAP Theorem
The CAP Theorem states that a distributed data store cannot simultaneously provide more than two of these three guarantees:

Consistency: Every read receives most recent write or an error
Availability: Every request receives a non-error response (but not guaranteed to contain most recent write)
Partition Tolerance: System operates despite arbitrary network issues

Database View of CAP Trade-offs

Consistency + Availability

System always responds with latest data
Every request gets a response
May not handle network partitions well

Consistency + Partition Tolerance

If system responds with data, it's always the latest
Otherwise, data request is dropped
Sacrifices some availability

Availability + Partition Tolerance

System always responds
May not provide absolute latest data
Most common choice for distributed systems

CAP in Reality
What it really means:

If you cannot limit fault numbers, requests can go to any server, and you insist on serving every request, then consistency is impossible

Common interpretation:

You must always sacrifice one: consistency, availability, or partition tolerance

NoSQL & Key-Value Databases
Distributed Databases and Concurrency Models
Pessimistic Concurrency (ACID)

Focuses on "data safety"
Assumes transactions need protection from other transactions
Conflicts prevented by locking resources until transaction completion
Uses both read and write locks
Analogy: Borrowing a library book - if you have it, no one else can use it
More details: How Databases Guarantee Isolation

Optimistic Concurrency

Transactions do not obtain locks on data for reading/writing
Assumes conflicts are unlikely to occur
Implementation: Add timestamp and version columns to tables

Read these when changing data
Check at transaction end if another transaction modified them

Works well for:

Low-conflict systems (backups, analytical DBs)
Read-heavy systems
Systems that can tolerate rollbacks and retries

Less efficient for high-conflict systems where locking may be preferable

NoSQL Overview

Term "NoSQL" first used in 1998 by Carlo Strozzi for a relational database without SQL
Modern meaning: "Not Only SQL" (sometimes interpreted as non-relational DBs)
Developed partly as a response to processing unstructured web-based data
Brief History of Non-Relational Databases

CAP Theorem
You can have 2, but not all 3, of the following:

Consistency: Every user has an identical view of data at any given instant
Availability: Database remains operational during failures
Partition Tolerance: Database maintains operations despite network failures between system segments

Note: Consistency in CAP differs from Consistency in ACID
CAP Trade-offs

Consistency + Availability: System always responds with latest data but may not handle network partitions
Consistency + Partition Tolerance: System responds with latest data or drops the request
Availability + Partition Tolerance: System always responds but may not provide the absolute latest data

BASE Model (ACID Alternative for Distributed Systems)

Basically Available: Data is available but might be inconsistent or changing
Soft State: System state may change without input due to eventual consistency

Data stores don't require write-consistency
Replicas don't require mutual consistency

Eventual Consistency: System will eventually become consistent when writes stop

Key-Value Databases
Core Design Principles

Simplicity

Extremely simple data model (key = value)
Supports basic CRUD operations and API creation
Much simpler than relational tables

Speed

Typically deployed as in-memory databases
O(1) retrieval operations using hash tables or similar structures

No complex queries or joins to slow things down

Scalability

Easy horizontal scaling by adding nodes
Uses eventual consistency in distributed environments

Use Cases
Data Science Applications

EDA/Experimentation Results Store

Intermediate results from data preprocessing
A/B testing results without production DB impact

Feature Store

Low-latency retrieval for model training and prediction

Model Monitoring

Store real-time performance metrics

Software Engineering Applications

Session Information Storage

Fast single-operation retrieval and storage

User Profiles & Preferences

Language, timezone, UI preferences

Shopping Cart Data

Cross-browser/device availability

Caching Layer

Front-end for disk-based databases

Redis (Remote Directory Server)
Overview

Open source, in-memory database
Sometimes called a "data structure store"
Primarily a KV store, but supports other models:

Graph, Spatial, Full Text Search, Vector, Time Series

Top-ranked key-value store according to db-engines.com

Key Features

In-memory but supports data durability through:

Disk snapshots at intervals
Append-only file journal for roll-forward recovery

Originally developed in 2009 in C++
Performance: >100,000 SET operations/second
Rich command set
Limitations: No complex data, no secondary indexes, lookup by key only

Data Types

Keys: Usually strings (can be any binary sequence)
Values:

Strings
Lists (linked lists)
Sets (unique unsorted string elements)
Sorted Sets
Hashes (string → string)
Geospatial data
JSON

Redis Databases

16 databases by default (numbered 0-15)
No other naming conventions
Interaction through commands or language libraries

Redis Data Types and Commands
Strings

Simplest data type: maps string to string
Can store text, serialized objects, binary arrays
Use Cases:

Caching HTML/CSS/JS fragments
Configuration/user settings
Token management
Page view counting
Rate limiting

String Commands

```
CopySET /path/to/resource 0
SET user:1 "John Doe"
GET /path/to/resource
EXISTS user:1
DEL user:1
KEYS user*
SELECT 5                # select database 5
INCR someValue          # increment by 1
INCRBY someValue 10     # increment by 10
DECR someValue          # decrement by 1
DECRBY someValue 5      # decrement by 5
SETNX key value         # set only if key doesn't exist
```

Hashes

Value is a collection of field-value pairs
Use Cases:

Represent basic objects/structures
Session information management
User/event tracking
Active session tracking

Hash Commands
CopyHSET bike:1 model Demios brand Ergonom price 1971
HGET bike:1 model
HGET bike:1 price
HGETALL bike:1
HMGET bike:1 model price weight
HINCRBY bike:1 price 100
Lists

Value is a linked list of string values
Use Cases:

Stacks and queues implementation
Message passing queues
Logging systems
Social media streams/feeds
Chat application message history
Batch processing task queues

List Commands
Copy# Queue operations
LPUSH bikes:repairs bike:1
LPUSH bikes:repairs bike:2
RPOP bikes:repairs

Stack operations
LPUSH bikes:repairs bike:1
LPUSH bikes:repairs bike:2
LPOP bikes:repairs

Other operations
LLEN mylist
LRANGE mylist 0 3   # elements from index 0 to 3
LRANGE mylist 0 0   # first element
LRANGE mylist -2 -1   # last two elements
Sets

Unordered collection of unique strings
Use Cases:

Track unique items (IP addresses)
Primitive relations
Access control lists
Social network friends/group membership
Supports set operations

Set Commands
CopySADD ds4300 "Mark"
SADD ds4300 "Sam"

```
SADD cs3200 "Nick"
SADD cs3200 "Sam"
SISMEMBER ds4300 "Mark"     # check membership
SCARD ds4300                # count elements
SINTER ds4300 cs3200        # intersection
SDIFF ds4300 cs3200         # difference
SREM ds4300 "Mark"          # remove element
SRANDMEMBER ds4300          # random member
```

JSON Type

Full support of JSON standard
Uses JSONPath syntax for parsing/navigating
Stored internally in binary tree structure for fast sub-element access

Redis Setup Guide

Pre-Requisites

Docker Desktop installed
JetBrains DataGrip installed

Step 1: Find the Redis Image

Open Docker Desktop
Use the built-in search to find the Redis image
Click "Run"

Step 2: Configure & Run the Container

Give the new container a name
Enter 6379 in the Host Port field (standard Redis port)
Click "Run"
Allow Docker time to download and start Redis

Step 3: Set up Data Source in DataGrip

Start DataGrip
Create a new Redis Data Source by either:

Clicking the "+" icon in the Database Explorer
Selecting "New" from the File menu

Step 4: Configure the Data Source

Give the data source a name
Install drivers if needed (a message will appear above "Test Connection" if required)
Test the connection to Redis
Click "OK" if connection test was successful

Notes

Redis can store various data types: strings, numbers, JSON objects, binary objects, etc.
If drivers aren't already installed, DataGrip will show a message above the "Test Connection" button

Redis-py Guide

Overview

Redis-py is the official Python client for Redis, maintained by Redis, Inc.

GitHub: redis/redis-py
Installation: pip install redis
Storage capability: Handles strings, numbers, JSON objects, and binary data

Connection
pythonCopyimport redis

```
# Connect to Redis server
redis_client = redis.Redis(
    host='localhost',  # For Docker: localhost or 127.0.0.1
    port=6379,         # Default Redis port
    db=2,              # Database number (0-15)
    decode_responses=True  # Converts byte responses to strings
)
```

Key Commands by Data Type
String Operations
pythonCopy# Basic operations

```
redis_client.set('clickCount:/abc', 0)
value = redis_client.get('clickCount:/abc')
redis_client.incr('clickCount:/abc')

# Multiple operations
redis_client.mset({'key1': 'val1', 'key2': 'val2', 'key3': 'val3'})
values = redis_client.mget('key1', 'key2', 'key3')  # Returns ['val1', 'val2',
'val3']
```

Available String Commands:

Write: set(), mset(), setex(), msetnx(), setnx()
Read: get(), mget(), getex(), getdel()
Numeric: incr(), decr(), incrby(), decrby()
Text: strlen(), append()

List Operations
pythonCopy# Create and populate list

```
redis_client.rpush('names', 'mark', 'sam', 'nick')
all_names = redis_client.lrange('names', 0, -1)  # Returns ['mark', 'sam',
'nick']
```

Available List Commands:

Left operations: lpush(), lpop()
Right operations: rpush(), rpop()
Management: lset(), lrem(), lrange(), llen(), lpos()

Hash Operations
pythonCopy# Create and populate hash

```
redis_client.hset('user-session:123', mapping={
    'first': 'Sam',
    'last': 'Uelle',
    'company': 'Redis',
    'age': 30
})

# Get all hash fields
user_data = redis_client.hgetall('user-session:123')
```

Available Hash Commands:

Basic: hset(), hget(), hgetall()
Management: hkeys(), hdel(), hexists(), hlen(), hstrlen()

Pipelines
Reduce network overhead by batching multiple commands:
pythonCopy# Create pipeline

```
pipe = redis_client.pipeline()
```

```
# Set multiple values in one execution
for i in range(5):
    pipe.set(f"seat:{i}", f"#{i}")
results = pipe.execute()  # [True, True, True, True, True]

# Chain commands
results =
redis_client.pipeline().get("seat:0").get("seat:3").get("seat:4").execute()
# Results: ['#0', '#3', '#4']
```

Redis in ML/Data Science

Redis serves as an essential component in ML architectures:

Feature stores
Vector databases
Model serving
Caching layers
Real-time processing

Resources

Full Redis Command List
Redis-py Documentation

Document Database

A Document Database is a non-relational database that stores data as structured documents, usually in JSON.
They are designed to be simple, flexible, and scalable.

What is JSON?

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate.
JSON is built on two structures:
1. A collection of name/value pairs (e.g., object, record, dictionary, hash table, associative array).
2. An ordered list of values (e.g., array, vector, list, sequence).
These two structures are supported by almost all modern programming languages, making JSON an ideal data interchange format.

JSON Syntax

Binary JSON? BSON
BSON (Binary JSON) is a binary-encoded serialization of a JSON-like document structure.
- Supports extended types like Date and BinaryData.
- Lightweight to minimize space overhead.
- Designed to be easily traversed, which is important for document databases.
- Efficient encoding and decoding.
- Supported by many programming languages.

XML (eXtensible Markup Language)
XML was the precursor to JSON as a data exchange format.
- XML + CSS is used to create web pages that separate content and formatting.
- XML is structurally similar to HTML, but the tag set is extensible.

XML-Related Tools/Technologies
- Xpath: A syntax for retrieving specific elements from an XML document.
- Xquery: A query language for XML documents (similar to SQL).
- DTD: A language to define the structure of an XML document.
- XSLT: A tool to transform XML into other formats, including HTML.

Why Document Databases?
Document databases solve the impedance mismatch problem between object-oriented programming (OOP) and how relational databases structure data. In OOP, inheritance and composition of types are used. However, saving complex objects in a relational database requires deconstructing them. Document databases allow the structure of a document to be self-describing, making them a natural fit for applications that use JSON/XML for data transport.

MongoDB
MongoDB started in 2007 after Doubleclick was acquired by Google, and three of its veterans realized the limitations of relational databases for serving >400,000 ads per second.
MongoDB was short for "Humongous Database."
MongoDB Atlas was released in 2016, offering document databases as a service.

MongoDB Structure
- Database
  - Collection A
  - Collection B
  - Collection C
    - Document 1
    - Document 2
    - Document 3

MongoDB Documents
- No predefined schema for documents.
- Every document in a collection can have a different schema.

Relational vs Mongo/Document DB
RDBMS vs MongoDB:
- Database: Database
- Table/View: Collection
- Row: Document
- Column: Field
- Index: Index
- Join: Embedded Document
- Foreign Key: Reference

MongoDB Features:
- Rich Query Support: Full support for CRUD operations.
- Indexing: Supports primary and secondary indexes on document fields.
- Replication: Supports replica sets with automatic failover.
- Load Balancing: Built-in load balancing.

MongoDB Versions:
- MongoDB Atlas: Fully managed MongoDB service (DBaaS).
- MongoDB Enterprise: Subscription-based, self-managed version.
- MongoDB Community: Source-available, free-to-use, self-managed version.

Interacting with MongoDB:
- mongosh: MongoDB Shell (CLI tool).
- MongoDB Compass: Free, open-source GUI for working with MongoDB.
- DataGrip and other 3rd-party tools: Libraries to interface with MongoDB for various languages (e.g., PyMongo for Python, Mongoose for JavaScript).

MongoDB Community Edition in Docker:
- Create a container and map the host:container port 27017.
- Provide an initial username and password for the superuser.

MongoDB Compass:
GUI tool for interacting with MongoDB. Download and install from the official website.

Load MFlix Sample Data Set:

1. Create a new database called mflix.
2. Download and unzip the mflix sample dataset.
3. Import JSON files for users, theaters, movies, and comments into new collections in the mflix database.

Creating a Database and Collection:
- Use mongosh to interact with MongoDB.
- Find queries in MongoDB are similar to SQL SELECT statements.

Example:
To select all users:
- `db.users.find()`

To filter users by name:
- `db.users.find({"name": "Davos Seaworth"})`

To filter movies released in 2010 and having a specific rating or genre:
- `db.movies.find({"year": 2010, $or: [{"awards.wins": {$gte: 5}}, {"genres": "Drama"}]})`

Count Documents in a Collection:
- To count documents, use `countDocuments()` method.

Example:
To count movies from 2010 with at least 5 awards or with a Drama genre:
- `db.movies.countDocuments({"year": 2010, $or: [{"awards.wins": {$gte: 5}}, {"genres": "Drama"}]})`

PyMongo
PyMongo is a Python library for interfacing with MongoDB instances.

PyMongo

PyMongo is a Python library for interfacing with MongoDB instances.

Getting a Database and Collection

```
from pymongo import MongoClient

client = MongoClient(
    'mongodb://user_name:pw@localhost:27017'
)

db = client['ds4300']  # or client.ds4300
collection = db['myCollection']  # or db.myCollection
```

Inserting a Single Document

```
db = client['ds4300']
collection = db['myCollection']

post = {
    "author": "Mark",
    "text": "MongoDB is Cool!",
    "tags": ["mongodb", "python"]
}

post_id = collection.insert_one(post).inserted_id
print(post_id)
```

Find all Movies from 2000

```
from bson.json_util import dumps
```

```
# Find all movies released in 2000
movies_2000 = db.movies.find({"year": 2000})

# Print results
print(dumps(movies_2000, indent=2))
```

Jupyter Time

- Activate your DS4300 conda or venv Python environment.
- Install pymongo with `pip install pymongo`.
- Install Jupyter Lab in your Python environment with `pip install jupyterlab`.
- Download and unzip this zip file — it contains 2 Jupyter Notebooks.
- In terminal, navigate to the folder where you unzipped the files, and run `jupyter lab`.

Graph Databases and Graph Theory

What is a Graph Database?

Data model based on the graph data structure
Composed of nodes (vertices) and edges (relationships)

Edges connect nodes
Each element is uniquely identified
Each can contain properties (e.g., name, occupation)

Supports graph-oriented operations:

Traversals
Shortest path algorithms
Many other specialized queries

Real-World Graph Applications

Social Networks

Social media platforms (Instagram, Facebook)
Modeling social interactions in psychology and sociology

The Web

A large graph of pages (nodes) connected by hyperlinks (edges)

Chemical and Biological Data

Systems biology, genetics
Chemical interaction relationships

Labeled Property Graph Model

Composed of node (vertex) objects and relationship (edge) objects
Labels: Used to mark nodes as part of a group
Properties: Key-value pair attributes that can exist on both nodes and relationships
Structure Rules:

Nodes with no relationships are permitted
Edges not connected to nodes are not allowed

Graph Example Components

2 Labels:

Person
Car

4 Relationship Types:

Drives
Owns
Lives_with
Married_to

Properties: Attributes attached to nodes and relationships

Paths in Graphs
A path is an ordered sequence of nodes connected by edges in which no nodes or edges are repeated.
Example:

Valid path: 1 → 2 → 6 → 5
Invalid path: 1 → 2 → 6 → 2 → 3 (node 2 repeats)

Graph Types and Properties
Connected vs. Disconnected

Connected: A path exists between any two nodes in the graph
Disconnected: Contains isolated components with no paths between them

Weighted vs. Unweighted

Weighted: Edges have a weight property (important for certain algorithms)
Unweighted: No weights assigned to edges

Directed vs. Undirected

Directed: Relationships (edges) define a specific start and end node
Undirected: Relationships have no direction

Cyclic vs. Acyclic

Cyclic: Graph contains at least one cycle
Acyclic: Graph contains no cycles

Sparse vs. Dense

Sparse: Relatively few edges compared to the maximum possible
Dense: Has many edges, approaching the maximum possible number

Trees
A tree is a special type of graph:

Connected
Acyclic
Undirected
Each node (except the root) has exactly one parent

Graph Algorithms Overview
Pathfinding Algorithms

Definition: Find the shortest path between nodes (fewest edges or lowest weight)
Use Case: Monitor network efficiency and resiliency using Average Shortest Path
Variants: Minimum spanning tree, cycle detection, max/min flow algorithms
Search Approaches:

BFS (Breadth-First Search): Explores all neighbors before moving deeper
DFS (Depth-First Search): Explores as far as possible along branches before backtracking

Centrality & Community Detection

Centrality: Identifies "important" nodes in a network (e.g., social network influencers)
Community Detection: Evaluates clustering/partitioning of nodes and their cohesion

Famous Graph Algorithms

Dijkstra's Algorithm: Single-source shortest path for positively weighted graphs
A Algorithm*: Enhanced Dijkstra's that uses heuristics to guide traversal
PageRank: Measures node importance based on incoming relationships and their sources

Neo4j

Type: Graph database system supporting transactional and analytical processing
Classification: NoSQL database with schema-optional design
Features:

Various indexing capabilities
ACID compliance
Distributed computing support

Similar Systems: Microsoft Cosmos DB, Amazon Neptune

This optimized format eliminates the unnecessary numbering (15-22), organizes content into logical sections, and uses consistent formatting that LLMs can easily process and understand.


```
{
 "cells": [
  {
   "cell_type": "markdown",
   "id": "02ead21f-dc3b-4536-aa57-f2ead5395348",
   "metadata": {},
   "source": [
    "# MongoDB Aggregation Examples"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "66da7d8d-4c4f-4c60-8e22-d0c023c35a0f",
   "metadata": {},
   "source": [
    "- Ensure you have pymongo installed before running cells in this notebook"
   ]
  },
```

```
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "5e29baf9-89a0-4618-9f91-2c8087128b7d",
   "metadata": {},
   "outputs": [],
   "source": [
    "import pymongo\n",
    "from bson.json_util import dumps\n",
    "import pprint\n",
    "\n",
    "#  -->  Update the URI with your username and password <--\n",
    "\n",
    "uri = \"mongodb://mark:abc123@localhost:27017/\"\n",
    "client = pymongo.MongoClient(uri)\n",
    "mflixdb = client.mflix\n",
    "demodb = client.demodb"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "1565e788-2c72-4ab6-a5b3-36e05334233b",
   "metadata": {},
   "source": [
    "## About Aggregates in PyMongo\n",
    "\n",
    "- Aggregation uses _pipelines_.\n",
    "- A **pipeline** is a sequence of stages through which documents proceed.
\n",
    "- Some of the different stages that can be used are:\n",
    "  - match\n",
    "  - project\n",
    "  - sort\n",
    "  - limit\n",
    "  - unwind\n",
    "  - group\n",
    "  - lookup"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "921429f5-3125-475e-b0af-a871b9a0799e",
   "metadata": {},
   "source": [
    "### $match"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "1a9f528f-5ffa-40fa-99b6-5a3bc36b3e33",
   "metadata": {
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "c = mflixdb.movies.aggregate([\n",
    "    {\"$match\": {\"year\": {\"$lte\": 1920}}},\n",
    "])\n",
    "\n",
    "print(dumps(c, indent=4))"
   ]
  },
  {
```

```
 "cell_type": "markdown",
 "id": "e180713d-92ec-4392-ba99-6c5a96775903",
 "metadata": {},
 "source": [
  "### match and project"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "id": "cc479ecc-7241-49d7-9cc2-5a57cd40bb5c",
 "metadata": {
  "scrolled": true
 },
 "outputs": [],
 "source": [
  "c = mflixdb.movies.aggregate([\n",
  "    {\"$match\": {\"year\": {\"$lte\": 1920}}},\n",
  "    {\"$project\": {\"_id\":0, \"title\": 1, \"cast\": 1}},\n",
  "])\n",
  "\n",
  "print(dumps(c, indent=4))"
 ]
},
{
 "cell_type": "markdown",
 "id": "7f7d7755-e9fa-4061-ac98-5f31b75201f2",
 "metadata": {},
 "source": [
  "### match project limit and sort"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "id": "d7e42c04-668e-4dd6-abf7-7a081c126c13",
 "metadata": {
  "scrolled": true
 },
 "outputs": [],
 "source": [
  "c = mflixdb.movies.aggregate([\n",
  "    {\"$match\": {\"year\": {\"$lte\": 1920}}},\n",
  "    {\"$sort\": {\"title\": 1}},\n",
  "    {\"$limit\": 5},\n",
  "    {\"$project\": {\"_id\":0, \"title\": 1, \"cast\": 1}},\n",
  "])\n",
  "\n",
  "print(dumps(c, indent=4))"
 ]
},
{
 "cell_type": "markdown",
 "id": "6e58c4a2-ff42-48a1-8493-aba643b7592d",
 "metadata": {},
 "source": [
  "### Unwind"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "id": "02ac7ecc-69ee-45cb-83ef-b271afaccbe1",
 "metadata": {
```

```
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "c = mflixdb.movies.aggregate([\n",
    "    {\"$match\": {\"year\": {\"$lte\": 1920}}},\n",
    "    {\"$sort\": {\"imdb.rating\": -1}},\n",
    "    {\"$limit\": 5},\n",
    "    {\"$unwind\": \"$cast\"},\n",
    "    {\"$project\": {\"_id\":0, \"title\": 1, \"cast\": 1, \"rating\":
\"$imdb.rating\"}},\n",
    "])\n",
    "\n",
    "print(dumps(c, indent=4))"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "4a2edddc-422e-4f67-8ddb-87343c5eb004",
   "metadata": {},
   "source": [
    "## Grouping"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "4e3b2ef1-681c-4cf4-baba-7d1399de1dd1",
   "metadata": {
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "# What is the average IMDB rating of all movies by year? sort the data by
year.\n",
    "\n",
    "c = mflixdb.movies.aggregate([\n",
    "    {\"$group\": {\"_id\": {\"release year\": \"$year\"}, \"Avg Rating\":
{\"$avg\": \"$imdb.rating\"}}},\n",
    "    {\"$sort\" : {\"_id\": 1}}\n",
    "    \n",
    "])\n",
    "print(dumps(c, indent = 2))"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "665047a1-87cb-422f-91ae-645714020a32",
   "metadata": {
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "# What is the average IMDB rating of all movies by year? sort the data by
avg rating in decreasing order.\n",
    "\n",
    "c = mflixdb.movies.aggregate([\n",
    "    {\"$group\": {\"_id\": {\"release year\": \"$year\"}, \"Avg Rating\":
{\"$avg\": \"$imdb.rating\"}}},\n",
    "    {\"$sort\" : {\"Avg Rating\": -1, \"_id\": 1}}\n",
    "    \n",
    "])\n",
    "print(dumps(c, indent = 2))"
```

```
   ]
  },
  {
   "cell_type": "markdown",
   "id": "f3bb3303-7cce-44f2-a9bb-7503ee2741e1",
   "metadata": {},
   "source": [
    "## Lookup"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "6f869bbe-701b-4fb3-980f-be9f1879b1ed",
   "metadata": {
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "data = demodb.customers.aggregate([\n",
    "    {\n",
    "        \"$lookup\": {\n",
    "            \"from\": \"orders\",\n",
    "            \"localField\": \"custid\",\n",
    "            \"foreignField\": \"custid\",\n",
    "            \"as\": \"orders\"\n",
    "        }\n",
    "    },\n",
    "    {\"$project\": {\"_id\": 0, \"address\": 0}}\n",
    "])\n",
    "print(dumps(data, indent = 2))"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "7e24db4a-38f0-4464-ab8b-19d675af29db",
   "metadata": {},
   "source": [
    "## Reformatting Queries"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "1d7f801c-05e1-4c05-baad-673c36302d20",
   "metadata": {},
   "outputs": [],
   "source": [
    "match = {\"$match\": {\"year\": {\"$lte\": 1920}}}\n",
    "limit = {\"$limit\": 5}\n",
    "project = {\"$project\": {\"_id\":0, \"title\": 1, \"cast\": 1, \"rating\": \"$imdb.rating\"}}\n",
    "\n",
    "agg = mflixdb.movies.aggregate([match, limit, project])\n",
    "print(dumps(agg, indent=2))"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "76e9276d-938d-45e3-92f4-f2daa42ce704",
   "metadata": {},
   "outputs": [],
   "source": []
```

```
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3 (ipykernel)",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.11.9"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 5
}


{
 "cells": [
  {
   "cell_type": "markdown",
   "id": "02ead21f-dc3b-4536-aa57-f2ead5395348",
   "metadata": {},
   "source": [
    "# MongoDB + PyMongo Example Queries"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "66da7d8d-4c4f-4c60-8e22-d0c023c35a0f",
   "metadata": {},
   "source": [
    "- Make sure your MongoDB container is running\n",
    "- Make sure you have pymongo installed before running cells in this
notebook.  If not, use `pip install pymongo`. "
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "5e29baf9-89a0-4618-9f91-2c8087128b7d",
   "metadata": {},
   "outputs": [],
   "source": [
    "import pymongo\n",
    "from bson.json_util import dumps\n",
    "\n",
    "#  -->  Update the URI with your username and password <--\n",
    "\n",
    "uri = \"mongodb://username:password@localhost:27017\"\n",
    "client = pymongo.MongoClient(uri)\n",
    "mflixdb = client.mflix"
   ]
  },
  {
```

```
   "cell_type": "code",
   "execution_count": null,
   "id": "e4ec02e0-7f92-40dc-a6db-c0bd0e2d5b32",
   "metadata": {},
   "outputs": [],
   "source": [
    "# Setup DemoDB with 2 collections\n",
    "demodb.customers.drop()\n",
    "demodb.orders.drop()\n",
    "\n",
    "customers = [\n",
    "   {\"custid\": \"C13\", \"name\": \"T. Cruise\", \"address\":
{ \"street\": \"201 Main St.\", \"city\": \"St. Louis, MO\", \"zipcode\":
\"63101\" }, \"rating\": 750 },\n",
    "   {\"custid\": \"C25\", \"name\": \"M. Streep\", \"address\":
{ \"street\": \"690 River St.\", \"city\": \"Hanover, MA\", \"zipcode\":
\"02340\" }, \"rating\": 690 },\n",
    "   {\"custid\": \"C31\", \"name\": \"B. Pitt\", \"address\": { \"street\":
\"360 Mountain Ave.\", \"city\": \"St. Louis, MO\", \"zipcode\": \"63101\" } },
\n",
    "   {\"custid\": \"C35\", \"name\": \"J. Roberts\", \"address\":
{ \"street\": \"420 Green St.\", \"city\": \"Boston, MA\", \"zipcode\":
\"02115\" }, \"rating\": 565 },\n",
    "   {\"custid\": \"C37\", \"name\": \"T. Hanks\", \"address\": { \"street\":
\"120 Harbor Blvd.\", \"city\": \"Boston, MA\", \"zipcode\": \"02115\" },
\"rating\": 750 },\n",
    "   {\"custid\": \"C41\", \"name\": \"R. Duvall\", \"address\":
{ \"street\": \"150 Market St.\", \"city\": \"St. Louis, MO\", \"zipcode\":
\"63101\" }, \"rating\": 640 },\n",
    "   {\"custid\": \"C47\", \"name\": \"S. Loren\", \"address\": { \"street\":
\"Via del Corso\", \"city\": \"Rome, Italy\" }, \"rating\": 625 }\n",
    "]\n",
    "\n",
    "orders = [\n",
    "   { \"orderno\": 1001, \"custid\": \"C41\", \"order_date\": \"2017-04-
29\", \"ship_date\": \"2017-05-03\", \"items\": [ { \"itemno\": 347, \"qty\": 5,
\"price\": 19.99 }, { \"itemno\": 193, \"qty\": 2, \"price\": 28.89 } ] },\n",
    "   { \"orderno\": 1002, \"custid\": \"C13\", \"order_date\": \"2017-05-
01\", \"ship_date\": \"2017-05-03\", \"items\": [ { \"itemno\": 460, \"qty\":
95, \"price\": 100.99 }, { \"itemno\": 680, \"qty\": 150, \"price\": 8.75 } ] },
\n",
    "   { \"orderno\": 1003, \"custid\": \"C31\", \"order_date\": \"2017-06-
15\", \"ship_date\": \"2017-06-16\", \"items\": [ { \"itemno\": 120, \"qty\": 2,
\"price\": 88.99 }, { \"itemno\": 460, \"qty\": 3, \"price\": 99.99 } ] },\n",
    "   { \"orderno\": 1004, \"custid\": \"C35\", \"order_date\": \"2017-07-
10\", \"ship_date\": \"2017-07-15\", \"items\": [ { \"itemno\": 680, \"qty\": 6,
\"price\": 9.99 }, { \"itemno\": 195, \"qty\": 4, \"price\": 35.00 } ] },\n",
    "   { \"orderno\": 1005, \"custid\": \"C37\", \"order_date\": \"2017-08-
30\", \"items\": [ { \"itemno\": 460, \"qty\": 2, \"price\": 99.98 },
{ \"itemno\": 347, \"qty\": 120, \"price\": 22.00 }, { \"itemno\": 780, \"qty\":
1, \"price\": 1500.00  }, { \"itemno\": 375, \"qty\": 2, \"price\":
149.98 } ] },\n",
    "   { \"orderno\": 1006, \"custid\": \"C41\", \"order_date\": \"2017-09-
02\", \"ship_date\": \"2017-09-04\", \"items\": [ { \"itemno\": 680, \"qty\":
51, \"price\": 25.98 }, { \"itemno\": 120, \"qty\": 65, \"price\": 85.00 },
{ \"itemno\": 460, \"qty\": 120, \"price\": 99.98 } ] },\n",
    "   { \"orderno\": 1007, \"custid\": \"C13\", \"order_date\": \"2017-09-
13\", \"ship_date\": \"2017-09-20\", \"items\": [ { \"itemno\": 185, \"qty\": 5,
\"price\": 21.99 }, { \"itemno\": 680, \"qty\": 1, \"price\": 20.50 } ] },\n",
    "   { \"orderno\": 1008, \"custid\": \"C13\", \"order_date\": \"2017-10-
13\", \"items\": [ { \"itemno\": 460, \"qty\": 20, \"price\": 99.99 } ] }\n",
    "]\n",
    "\n",
    "demodb.customers.insert_many(customers)\n",
```

```
    "demodb.orders.insert_many(orders)\n",
    "\n",
    "numCustomers = demodb.customers.count_documents({})\n",
    "numOrders = demodb.orders.count_documents({})\n",
    "\n",
    "print(f'There are {numCustomers} customers and {numOrders} orders')"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "4b977e9f-6cc5-4afb-91ed-fc7eec781c4e",
   "metadata": {
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "# The key (_id) attribute is automatically returned unless you explicitly
say to remove it. \n",
    "\n",
    "# SELECT name, rating FROM customers\n",
    "data = demodb.customers.find({}, {\"name\":1, \"rating\":1})\n",
    "print(dumps(data, indent=2))"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "38d4b619-5989-4962-b3e6-ec1893095d72",
   "metadata": {
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "# Now without the _id field. \n",
    "\n",
    "# SELECT name, rating FROM customers\n",
    "data = demodb.customers.find({}, {\"name\":1, \"rating\":1, \"_id\":0})\n",
    "print(dumps(data, indent=2))"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "1be79501-7f8b-43af-a955-9e9f95f842f2",
   "metadata": {},
   "source": [
    "### All fields EXCEPT specific ones returned"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "48385e0f-fe32-4abd-91a0-ef8e732046c3",
   "metadata": {
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "# For every customer, return all fields except _id and address.\n",
    "\n",
    "data = demodb.customers.find({}, {\"_id\": 0, \"address\": 0})\n",
    "print(dumps(data, indent=2))"
   ]
  },
```

```
  {
   "cell_type": "markdown",
   "id": "d929a632-57de-4de5-8049-6dada9027e6b",
   "metadata": {},
   "source": [
    "## Equivalent to SQL LIKE operator"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "84e41240-5186-41f1-ba94-3935e333e255",
   "metadata": {},
   "outputs": [],
   "source": [
    "# SELECT name, rating FROM customers WHERE name LIKE 'T%'\n",
    "\n",
    "# Regular Expression Explanation:\n",
    "  # ^ - match beginning of line\n",
    "  # T - match literal character T (at the beginning of the line in this
case)\n",
    "  # . - match any single character except newline\n",
    "  # * - match zero or more occurrences of the previous character (the . in
this case)\n",
    "\n",
    "data = demodb.customers.find({\"name\": {\"$regex\": \"^T.*\"}}, {\"_id\":
0, \"name\": 1, \"rating\":1})\n",
    "print(dumps(data, indent=2))"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "95c0ac04-ab9d-4a34-850a-16137cd1fc1d",
   "metadata": {},
   "source": [
    "## Sorting and limiting "
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "e07bf0b9-10f1-4fae-97f0-19ff2afda9bf",
   "metadata": {},
   "outputs": [],
   "source": [
    "# SELECT name, rating FROM customers ORDER BY rating LIMIT 2\n",
    "\n",
    "data = demodb.customers.find( { }, {\"_id\": 0, \"name\": 1,
\"rating\":1} ).sort(\"rating\").limit(2)\n",
    "print(dumps(data, indent=2))"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "08236f13-3f94-47da-bd19-f931c59039e4",
   "metadata": {},
   "outputs": [],
   "source": [
    "# Same as above, but sorting in DESC order\n",
    "\n",
    "# SELECT name, rating FROM customers ORDER BY rating DESC LIMIT 2\n",
    "\n",
    "data = demodb.customers.find( { }, {\"_id\": 0, \"name\": 1,
```

```
\"rating\":1} ).sort(\"rating\", -1).limit(2)\n",
    "print(dumps(data, indent=2))"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "7ae8e685-34d5-49b3-af1e-5a0f5ed6186c",
   "metadata": {},
   "outputs": [],
   "source": [
    "# Providing 2 sort keys... \n",
    "\n",
    "data = demodb.customers.find( { }, {\"_id\": 0, \"name\": 1,
\"rating\":1} ).sort({\"rating\": -1, \"name\": 1}).limit(2)\n",
    "print(dumps(data, indent=2))"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "143f4349-d0cf-4812-a27c-f627f709d8ae",
   "metadata": {},
   "source": [
    "# Your Turn with mflix DB"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "3e6cbf79-7813-4df0-8386-00b4c133ba0a",
   "metadata": {},
   "source": [
    "## Question 1"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 16,
   "id": "ca28ccf2-bff1-43be-a1b6-2c7f735e4c3b",
   "metadata": {},
   "outputs": [],
   "source": [
    "# How many Users are there in the mflix database? How many movies?\n"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "e520f1c3-f9bf-4668-8d84-7ef605c5bd90",
   "metadata": {},
   "source": [
    "## Question 2"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "224682d3-5448-4fcd-83b0-b4938b083b24",
   "metadata": {},
   "outputs": [],
   "source": [
    "# Which movies have a rating of “TV-G”? Only return the Title and Year.\n",
    "\n"
   ]
  },
  {
```

```
   "cell_type": "markdown",
   "id": "2a86daef-c330-493d-92d5-38517d7c4381",
   "metadata": {},
   "source": [
    "## Question 3"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "566f4500-2eb5-4190-abdf-bfb9228d5c3f",
   "metadata": {
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "# Which movies have a runtime of less than 20 minutes?  Only return the
title and runtime of each movie. \n",
    "\n"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "05d0bfeb-eb46-4fe0-a257-54b632f1c34f",
   "metadata": {},
   "source": [
    "## Question 4"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "86821f3e-a95e-40fd-a8bb-3afb7078cc6b",
   "metadata": {},
   "outputs": [],
   "source": [
    "# How many theaters are in MN or MA?\n",
    "\n",
    "
"
   ]
  },
  {
   "cell_type": "markdown",
   "id": "d3c90cac-74bc-4b69-9a08-b8118bf97bfd",
   "metadata": {},
   "source": [
    "## Question 5"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "03c20e7f-2d72-47c5-b104-083cca731cb5",
   "metadata": {
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "# Give the names of all movies that have no comments yet. Make sure the
names are in alphabetical order. \n",
    "\n"
   ]
  },
```

```
  {
   "cell_type": "markdown",
   "id": "dd98ae0a-0d11-4d33-a5a8-05f6a7da052e",
   "metadata": {},
   "source": [
    "## Question 6"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "2c5e0923-d2ec-4e86-a85f-57c4ef523b6f",
   "metadata": {
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "# Return a list of movie titles and all actors from any movie with a title
that contains the word 'Four'. \n",
    "# Sort the list by title. \n"
   ]
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3 (ipykernel)",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.11.9"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 5
}
```